华中科技大学

大数据管理实验报告

姓　　名：　　李嘉鹏

学　　院：　　计算机科学与技术学院

专　　业：　　数据科学与大数据技术

班　　级：　　大数据 2101 班

学　　号：　　U202115652

分数	
教师签名	

2023 年 11 月 4 日

教师评分页

子目标	子目标评分
1	
2	
3	
4	
5	
6	
7	
总分	

目　　录

1　课程任务概述 1

2　Mysql for Json 实验 2

2.1 任务要求 2

2.2 完成过程 2

2.3 任务小结 5

3　MongoDB 实验 7

3.1 任务要求 7

3.2 完成过程 7

3.3 任务小结 11

4　Neo4j 实验 12

4.1 任务要求 12

4.2 完成过程 12

4.3 任务小结 19

5　多数据库交互应用实验 20

5.1 任务要求 20

5.2 完成过程 20

5.3 任务小结 21

6　不同类型数据库 MVCC 多版本并发控制对比实验 22

6.1 任务要求 22

6.2 完成过程 22

6.3 任务小结 25

7　课程总结 26

1　课程任务概述

本学期的大数据管理实验课主要分为五个部分，分别涉及 MySQL for JSON、MongoDB、Neo4j 三类不同数据库的使用，以及多数据库交互应用、不同类型数据库 MVCC 多版本并发控制对比实验。通过使用 Yelp 数据集完成各类实验任务，可以体验搭建云环境上集群并部署数据管理软件，实施多种类型数据的管理和操作，并进一步理解大数据系统的典型系统架构、数据组织与存储策略、查询处理优化、分布式事务处理等技术。

对于实验一（MySQL for JSON 实验），涉及到关系数据模型查询语言。首先使用 MySQL for JSON 的语法完成一些 JSON 的基本查询，返回结果要求必须为 JSON 格式。然后对数据集实现一些简单的增（insert）、删（remove）、改（update）操作。最后使用聚合函数和实用函数（例如模式匹配函数）完成一些更复杂的查询，并使用 explain 观察数据库在执行排序时的内存开销。

对于实验二（MongoDB 实验），主要是对文档模型的查询。首先完成一些条件查询任务，并使用 explain 查看查询语句的执行计划，给出物理优化手段以提高查询性能。随后使用聚合与索引完成一些高级查询。最后还设置了理论课上讲过的 MapReduce 任务，即通过 map 和 reduce 两个步骤实现平均分的计算。

对于实验三（Neo4j 实验），核心在于图模型与 Cypher 语言。总体上还是完成一些基础语法查询，并使用 PROFILE 查看执行计划，提出合理的物理优化策略。除此之外还需要在查询过程中引入索引，对比分析索引对查询、插入、删除等操作带来的影响。

对于实验四（多数据库交互应用实验），主要是在两个数据库之间完成，例如首先使用 Neo4j 导出结果集，再将其导入 MongoDB 进行下一步查询。进一步，还可以将 MongoDB 得到的第二步结果集再次导入 Neo4j 中完成后续操作。

对于实验五（不同类型数据库 MVCC 多版本并发控制对比实验），需要分别在 MySQL 和 MongoDB 两种数据库下体验 MVCC 并发控制特性，自行构造用户多写多读样例，观察现象并进行对比分析。

2　Mysql for Json 实验

2.1 任务要求

本任务首先需要使用 MySQL for JSON 的语法完成一些 JSON 的基本查询，返回结果要求必须为 JSON 格式。然后对数据集实现一些简单的增（insert）、删（remove）、改（update）操作。最后使用聚合函数和实用函数（例如模式匹配 REGEXP_LIKE()函数）完成一些更复杂的查询，并使用 explain 观察数据库在执行排序时的内存开销。

2.2 完成过程

2.2.1 子任务 1-11

题目要求：在 user 表中查询 funny 大于 2000 且平均评分大于 4.0 的用户，返回他们的名字、平均评分以及按 json 数组形式表示的 funny, useful, cool 三者的和，限制 10 条；尝试按平均评分降序排序，并使用 explain 查看排序的开销，与第一题的排序情况做对比，主要关注"rows_examined_per_scan"和"cost_info"。

分析与实现：这道题涉及到多个不同的语法，包括将表的属性投影出来（使用'$.xxx'）并给出新的名称、通过 JSON_ARRAY 将用户的 funny, useful, cool 和三者之和表示为 json 数组、通过 order by 实现对结果按平均评分降序排序等。代码如下。

```
select
user_info->'$.name' as name,
user_info->'$.average_stars' as avg_stars,
JSON_ARRAY(user_info->'$.funny', user_info->'$.useful', user_info->'$.cool',
user_info->'$.funny'+user_info->'$.useful'+user_info->'$.cool') as '[funny,useful,cool,sum]'
from user
where user_info->'$.funny'>2000 and user_info->'$.average_stars'>4.0
order by user_info->'$.average_stars' desc
limit 10;
```

而第 1 题的代码更简单直接，为了后续进行查询开销的比较在此列出，如下所示。

```
select *
from business
where business_info->'$.city'="Tampa"
order by business_info->'$.review_count' desc
limit 10;
```

在上述代码前面加上“explain format = json”即可查看以 json 格式导出的排序执行开销，如图 2.1、图 2.2 所示。

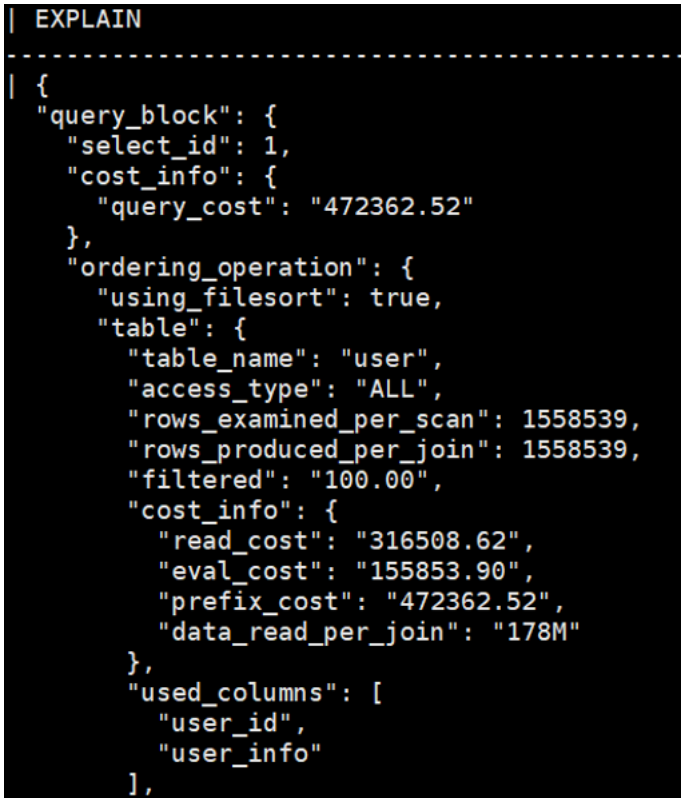


图 2.1　Mysql for Json 第 11 题执行开销（以 json 格式表示）

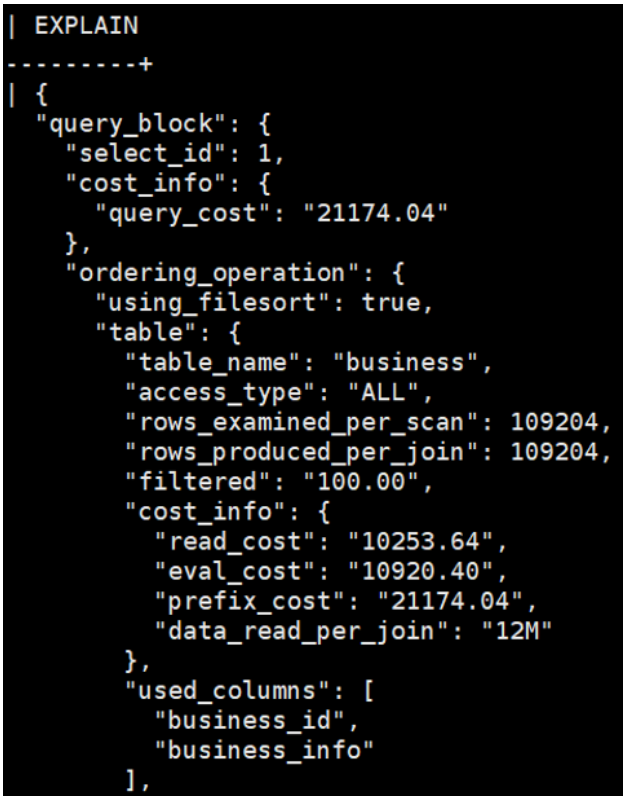


图 2.2　Mysql for Json 第 1 题执行开销（以 json 格式表示）

从上面可以看出两个关键信息：rows_examined_per_scan 指每次扫描过程中

访问的行数，此题中 rows_examined_per_scan=1558539，相比之下第 1 题的排序过程中 rows_examined_per_scan=109204，比前者小了一个数量级。这是因为第 11 题的查询语句中 WHERE 条件涉及到两个 JSON 字段的值，导致需要扫描更多的行来找到符合条件的数据。而且该查询需要返回多个 JSON 字段的值，也会增加查询涉及的数据量。而在第 1 题中，查询语句中 WHERE 条件只涉及一个 JSON 字段的值，且排序涉及的行数非常小，因此需要扫描的行数相对较小。

除此之外，还可以重点关注 cost_info 指标，这是指执行查询的相关成本估算信息。它包含四个指标，含义分别是：

①read_cost：访问数据行的成本估算值，表示在执行查询时预计会读取的数据块数量。

②eval_cost：计算过滤条件和合并结果集的成本估算值，也就是查询执行过程中需要进行的逻辑计算量。

③prefix_cost：前缀索引扫描的成本估算值。一般等于访问前缀索引的成本加上根据前缀索引返回的数据行的成本。

④data_read_per_join：执行 Join 操作时需要读取的数据量估算值，这个值等于连接中 Join 操作涉及的每个表的行数乘以该表的行大小。

第 11 题和第 1 题在这些指标上的排序开销如表 2.1 所示。

表 2.1　Mysql for Json 第 11 题和第 1 题排序开销对比

指标	*rows_examined_per_scan*	*cost_info*			
		read_cost	**eval_cost**	**prefix_cost**	**data_read_per_join**
11 题	1558539	316508.62	155853.90	472362.52	178M
1 题	109204	10253.64	10920.40	21174.04	12M

综合上面两个指标的差异，可以发现本题在所有查询开销指标上都显著大于第 1 题。可见如果在查询中 JSON 数据的上嵌套类型更多、层级更深、查询条件更复杂，则会需要扫描更多行来满足查询要求，从而查询效率会受到影响。高查询复杂度会导致高执行开销。

2.2.2 子任务 1-13

题目要求：查询被评论数前三的商户，使用 JSON_TABLE()导出他们的名字、被评论数、是否在星期二营业（即"hours"中有"Tuesday"的键值对，是返回 1，不是返回 0），和一周所有的营业时段（不考虑顺序，一个时段就对应一行，对每个商户，从 1 开始对这些时段递增编号），最后按商户名字升序排序。

分析与实现：这是一个具有多重限制的复杂查询，可以一个个写出查询条件对应的语句，并按照合适的先后顺序进行排布。

由于只要求查询被评论数前三的用户，因此可以首先做一个子查询，筛选出top3 的 business，将其保存为新的集合 sub。然后在子查询结果 sub 的基础上，使用 json_table 函数对“hours”列进行解析，将其列拆分为名为 time_slot 的子列，代表一周内每一天的营业时间段。对于“是否在星期二营业”这个值，可以使用“case when…then…else”实现一个分支结构，then 前面的判断条件满足时输出 0，否则输出 1。最后，通过在子查询和 json_table 的结果之间完成内连接即可得到最后的答案。这里通过“on 1=1”实现，因为没有特定的连接条件。

本题的代码如下。

```
select
    sub.business_name,
    sub.review_count as business_review_count,
    case when sub.hours->'$.Tuesday' is null then 0 else 1 end as business_open_on_Tuesday,
    jt.time_slot
from
    (select
        business_info->'$.name' as business_name,
        business_info->'$.review_count' as review_count,
        business_info->'$.hours' as hours
     from business
     order by business_info->'$.review_count' desc
     limit 3) as sub
join json_table(
        sub.hours,
        "$.*" columns (
            time_slot VARCHAR(255) PATH '$')
    ) as jt on 1=1
order by sub.business_name;
```

2.3 任务小结

在 Mysql for Json 实验中相对来说还算完成的比较顺利，开始时按照华为平台部署手册在云服务器上配置 Mysql 环境和下载数据集都没有问题，主要是在熟悉 Mysql for Json 查询语法上花了一些时间。

实验中遇到的一大问题是第 11 题进行排序操作时，由于数据集本身较大且数据库本身的缓存池不够大，因此在排序时报错“out of sort memory”，如图 2.3 所示。

```
mysql> select user_info->'$.name' as name, user_info->'$.average_stars' as avg_stars, JSON
_ARRAY(user_info->'$.funny', user_info->'$.useful', user_info->'$.cool', user_info->'$.fun
ny'+user_info->'$.useful'+user_info->'$.cool') as '[funny,useful,cool,sum]' from user wher
e user_info->'$.funny'>2000 and user_info->'$.average_stars'>4.0 order by user_info->'$.av
g_stars' limit 10;
ERROR 1038 (HY001): Out of sort memory, consider increasing server sort buffer size
```

图 2.3　Mysql for Json 第 11 题排序时报错

在网上查找相关方案，发现只需要手动修改缓存池大小即可。使用“set sort_buffer_size = 5000000”扩大其大小，并再次输入查询语句，发现可以正常排序并输出结果了，如图 2.4 所示。

```
mysql> set sort_buffer_size=5000000;
Query OK, 0 rows affected (0.00 sec)

mysql> select user_info->'$.name' as name, user_info->'$.average_stars' as avg_stars, JSON
_ARRAY(user_info->'$.funny', user_info->'$.useful', user_info->'$.cool', user_info->'$.fun
ny'+user_info->'$.useful'+user_info->'$.cool') as '[funny,useful,cool,sum]' from user wher
e user_info->'$.funny'>2000 and user_info->'$.average_stars'>4.0 order by user_info->'$.av
erage_stars' desc limit 10;
+-----------+-----------+--------------------------------+
| name      | avg_stars | [funny,useful,cool,sum]        |
+-----------+-----------+--------------------------------+
| "Markus"  | 5.0       | [7230, 8389, 8364, 23983.0]    |
| "Plan"    | 4.96      | [2303, 2836, 2768, 7907.0]     |
| "Sridhar" | 4.95      | [4316, 9937, 9196, 23449.0]    |
| "Stacy"   | 4.87      | [2453, 3137, 3072, 8662.0]     |
| "Susan"   | 4.83      | [36153, 65261, 62907, 164321.0] |
| "Eliza"   | 4.81      | [6324, 9861, 9201, 25386.0]    |
| "Alicia"  | 4.75      | [4016, 6361, 5000, 15377.0]    |
| "Noemi"   | 4.75      | [2890, 5715, 5806, 14411.0]    |
| "Robbie"  | 4.75      | [6600, 9579, 8314, 24493.0]    |
| "Linda"   | 4.74      | [2850, 3584, 3569, 10003.0]    |
+-----------+-----------+--------------------------------+
10 rows in set (32.72 sec)
```

图 2.4　手动修改缓存池大小，成功输出排序结果

通过本次实验，我掌握了 Mysql for JSON 的基本语法和功能，能灵活地处理数据集并做出特定的修改，以及使用 explain 函数查看查询计划开销，并尝试解读其背后的含义。在上学期的数据库实验课中，我已经对 Mysql 的使用有了初步了解，而此次任务更加侧重于对 json 类型数据的操作，使我更加深入地体会到这一数据库的广泛用途。

3 MongoDB 实验

3.1 任务要求

本实验首先需要使用 MongoDB 完成一些条件查询任务，使用 explain 查看查询语句的执行计划，了解 MongoDB 对集函数的执行方式，并给出物理优化手段以提高查询性能。随后使用聚合与索引完成一些高级查询。最后还设置了理论课上讲过的 MapReduce 任务，即通过 map 和 reduce 两个步骤实现平均分的计算。

3.2 完成过程

3.2.1 子任务 2-9

题目要求：使用 explain 函数查看查询语句 db.business.find({business_id: "5JucpCfHZltJh5r1JabjDg"})的执行计划，了解该查询的执行计划及查询执行时间，给出物理优化手段以提高查询性能，通过优化前后的性能对比展现优化程度。

分析与实现：使用 explain 查看执行计划，需要注意在 MongoDB 中 explain 需要放在查询语句的最后面。代码如下。

```
db.business.find({ business_id: "5JucpCfHZltJh5r1JabjDg" }).explain("executionStats")
```

该查询的执行计划如图 3.1 所示。从图中可以重点关注的关键词包括 parsedQuery 和 winningPlan。

对于 winningPlan，这是 MongoDB 优化器决策出的用于执行查询的最优执行计划。在这个查询语句中，优化器会尝试多个不同的执行计划，并选出成本和实际表现最优的计划作为最终的方案。此时 winningPlan 中的 stage 字段为“COLLSCAN”，代表在查询中遍历了整个集合，并一一比对判断是否满足查询条件，显然效率较低。

对于 parsedQuery，这是 MongoDB 查询解析器生成的 MongoDB 查询语句表达式。由于本题查询非常简单，只需要根据 business_id 查找相对应的 business，因此解析出来的查询表达式和原始语句没有差别。

图 3.1　MongoDB 第 9 题查询计划（优化前）

考虑使用索引作为物理优化手段。由于查询条件是关于 business_id 的，因此可以直接为 business_id 建立索引：

```
db.business.createIndex({'business_id':1})
```

如图 3.2 所示，在 business_id 上的索引已经成功建立。

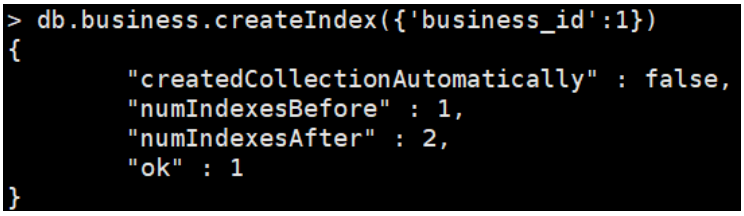


图 3.2　在 business_id 上建立索引

再次使用 explain 查看优化后的执行计划，如图 3.3 所示。可以发现此时 winningPlan 中的 stage 字段变为“FETCH”，代表在扫描索引后根据 id 从物理存储中检索出完整的数据，这是使用索引进行查询的必要步骤。同时，在 inputStage 字段中也出现了 stage 字段，其值为“IXSCAN”，这是索引扫描的一种类型，通过索引键值或索引范围的有序扫描来快速定位满足查询条件的数据。

```
> db.business.find({ business_id: "5JucpCfHZltJh5r1JabjDg" }).explain("executionStats")
{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "yelp.business",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "business_id" : {
                                "$eq" : "5JucpCfHZltJh5r1JabjDg"
                        }
                },
                "winningPlan" : {
                        "stage" : "FETCH",
                        "inputStage" : {
                                "stage" : "IXSCAN",
                                "keyPattern" : {
                                        "business_id" : 1
                                },
                                "indexName" : "business_id_1",
                                "isMultiKey" : false,
                                "multiKeyPaths" : {
                                        "business_id" : [ ]
                                },
                                "isUnique" : false,
                                "isSparse" : false,
                                "isPartial" : false,
                                "indexVersion" : 2,
                                "direction" : "forward",
                                "indexBounds" : {
                                        "business_id" : [
                                                "[\"5JucpCfHZltJh5r1JabjDg\", \"5JucpCfHZltJh5r1JabjDg\"]"
                                        ]
                                }
                        }
                },
                "rejectedPlans" : [ ]
        },
        "executionStats" : {
                "executionSuccess" : true,
                "nReturned" : 1,
                "executionTimeMillis" : 0,
                "totalKeysExamined" : 1,
                "totalDocsExamined" : 1,
                "executionStages" : {
                        "stage" : "FETCH",
                        "nReturned" : 1,
                        "executionTimeMillisEstimate" : 0,
                        "works" : 2,
                        "advanced" : 1,
                        "needTime" : 0,
                        "needYield" : 0,
                        "saveState" : 0,
                        "restoreState" : 0,
```

图 3.3　MongoDB 第 9 题查询计划（优化后）

下面对优化前后的执行开销进行比较。对于预计工作量（executionStats 中的 works 字段），优化前该字段的值为 192609，而在引入索引后该值变为 2，相比优化前大大降低。对于查询时间（executionTimeMillis）而言，优化前和优化后分别为 64 和 0，如图 3.2 和图 3.3 所示。

可见，索引扫描有助于提高查询性能，因为它避免了全表遍历扫描，而是通过索引的键来进行精确定位，完成快速的定位和搜索。

3.2.2 子任务 2-15

题目要求：使用 map reduce 计算每个商家的评价的平均分（在 Subreview 集合上做），不能直接使用聚合函数。

分析与实现：MapReduce 分为两个阶段：①在 map 阶段，需要以 business_id 作为键，抽取出关于它的每一条评价，并记录评价所对应的 stars 值，传递给 reduce 阶段进行统计；②在 reduce 阶段，可以根据 map 阶段记录的特定 business_id 的每一条评价，计算出其获得的 stars 总数量以及评价总数量。

最后，通过 reduce 阶段的统计结果即可进一步计算出其 stars 的平均值，也就是评价的平均分，其值等于该 business_id 获得的 stars 总数量除以评价总数量。整体的代码如下。

```
db.business.mapReduce(
function(){
emit(this.business_id, {stars: this.stars, count:1});},
function(key, values){
var totalStars=0;
var totalCount=0;
values.forEach(function(value) {
totalStars+=value.stars;
totalCount+=value.count;});
return {stars:totalStars, count:totalCount};},
{
out:"test_map_reduce",
finalize: function(key, reducedValue){
var avgStars = reducedValue.stars / reducedValue.count;
return { count: reducedValue.count, stars: reducedValue.stars, avg: avgStars };}})
```

使用 MapReduce 完成本题任务的流程图如图 3.4 所示。MongoDB 的 MapReduce 可以利用分布式计算环境，将计算任务分发到多个节点上进行并行处理。这样可以在较短的时间内完成大量数据的计算，进一步提高性能，对大规模数据集有很大的优势。

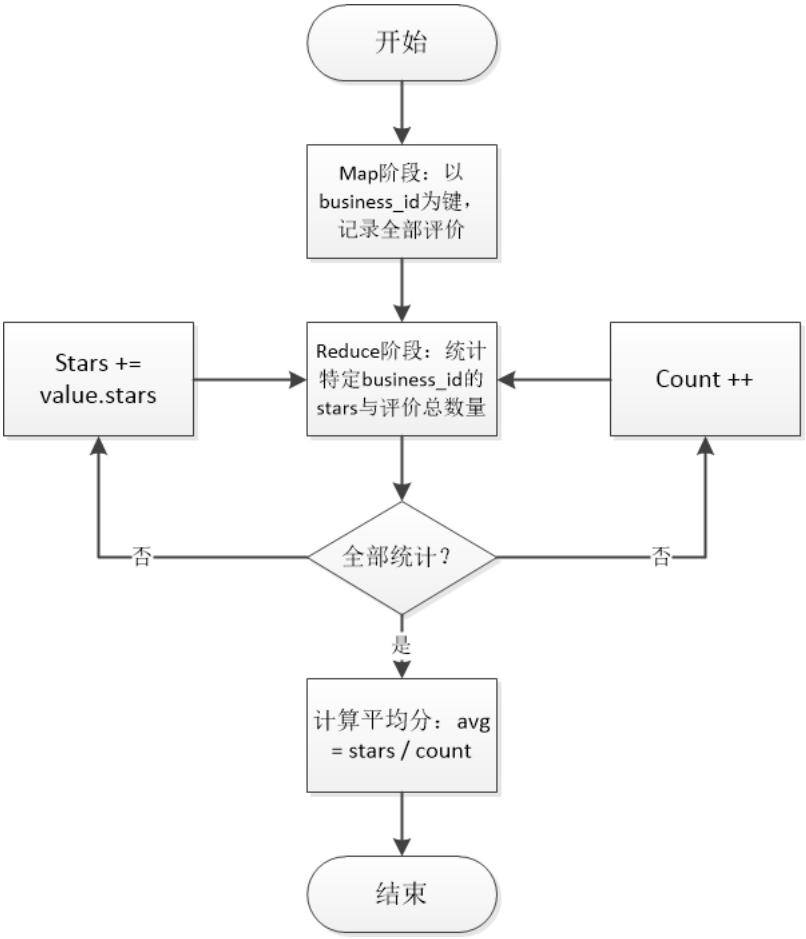


图 3.4　MongoDB 第 15 题 MapReduce 算法流程

3.3 任务小结

在 MongoDB 的实验中，首先一大难点在于习惯 Mongo 的查询语法。由于前一章 Mysql 的语法在上学期数据库系统原理实验课中已经比较熟悉，但是 Mongo 只是在理论课上介绍过，之前并没有写过相关的代码，因此刚开始上手时比较不适应。

这一节的重点在于我了解了索引对查询的帮助，尤其是在数据条数很多的情况下，在合适的数据属性上添加索引可以大大降低查询开销。

我还学习了 MapReduce 的实现方式，也就是在 Map 阶段将数据拆分成小的数据项，将输入的键值对映射为中间键值对，然后传递到 Reduce 阶段，对组内的值进行聚合和计算，最后输出结果。在 MongoDB 的分布式计算环境中，MapReduce 可以在多个节点上并行处理，提高数据处理效率。

4 Neo4j 实验

4.1 任务要求

Neo4j 是图数据库，核心在于构建图模型与使用 Cypher 语言。本节任务首先需要完成一些基础语法查询，并使用 PROFILE 查看执行计划，提出合理的物理优化策略。除此之外还需要在查询过程中引入索引，对比分析索引对查询、插入、删除等操作带来的影响。

4.2 完成过程

4.2.1 子任务 3-8

题目要求：查询 businessid 是 tyjquHslrAuF5EUejbPfrw 商家包含的种类数，并使用 PROFILE 查看执行计划，进行说明。

分析与实现：首先查询与商家节点（BusinessNode）有关的种类节点（CategoryNode）的数量，二者之间的关系可以表示为：

```
(:BusinessNode)-[:IN_CATEGORY]->(CategoryNode)
```

也就是说商家节点有一条有向边连接到种类节点，有向边的标签为“IN_CATEGOTY”。据此可写出查询代码，如下所示。

```
MATCH
(:BusinessNode {businessid:'tyjquHslrAuF5EUejbPfrw'})-[:IN_CATEGORY]->(c:CategoryNode)
RETURN count(c)
```

在查询语句最前面添加 PROFILE 查看执行计划，如图 4.1 所示。

可以看出，首先在 NodeIndexSeek 阶段查询了 business_id 匹配的商家节点，访问开销很小（仅为 2，这是由于在 3.2.1 节添加了索引），最后查询到 1 个满足条件的数据项（row）。在 Expand 和 Filter 阶段，数据库根据给定的关系查询到商家节点指向的 4 个种类节点，并通过 EagerAggregation 聚合出种类节点的数量，最后输出到 Result 中。

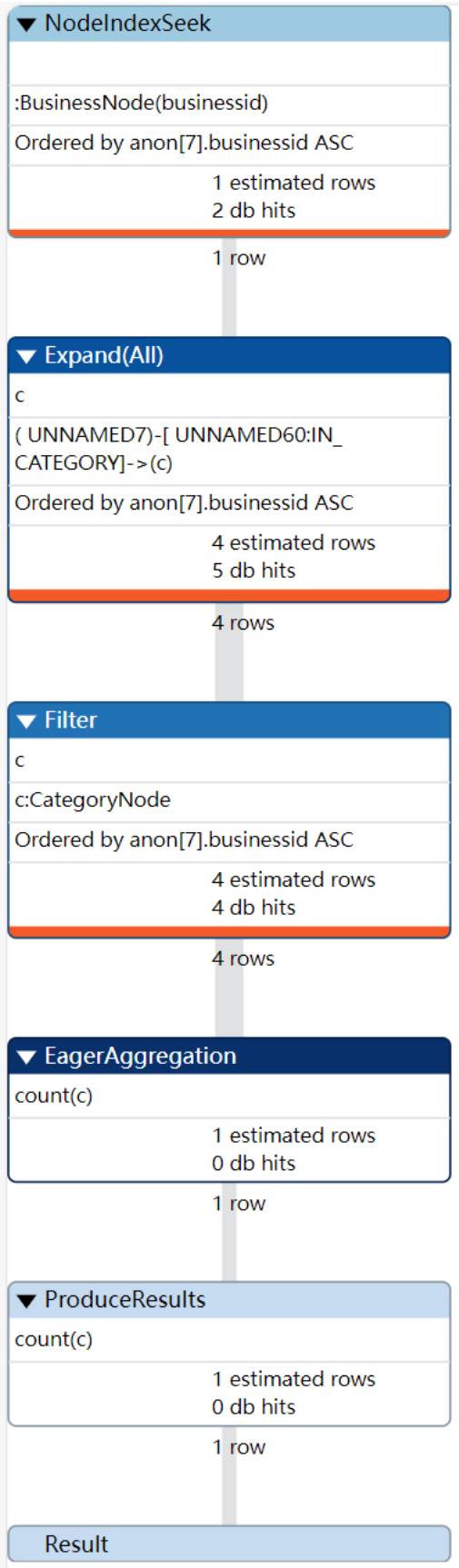


图 4.1　Neo4j 第 8 题查询执行计划

4.2.2 子任务 3-16

题目要求：体会建立索引对查询带来的性能提升，但会导致插入、删除等操作变慢（需要额外维护索引代价）。

分析与实现：为了不干扰原始数据库，我在这里为用户节点（UserNode）新增一个属性 flag 以完成本题。不妨设 flag 的值等于用户的 fans。

```
MATCH (user:UserNode)    WHERE toInteger(user.fans) > 400    SET user.flag = user.fans
```

这里在创建属性 flag 时需要对它的值做出限定（例如大于 400），如果不这样做会导致创建太多而溢出内存。接下来对 UserNode 的 flag 属性执行查询（flag>8000）、创建（flag=10000）、更新（flag=456 to flag=3456）、删除（flag>4000）操作。

查询操作的代码如下：

```
MATCH (user:UserNode)    WHERE toInteger(user.flag) > 8000    RETURN user
```

查询结果如图 4.2 所示，查询操作所用的时间如图 4.3 所示。

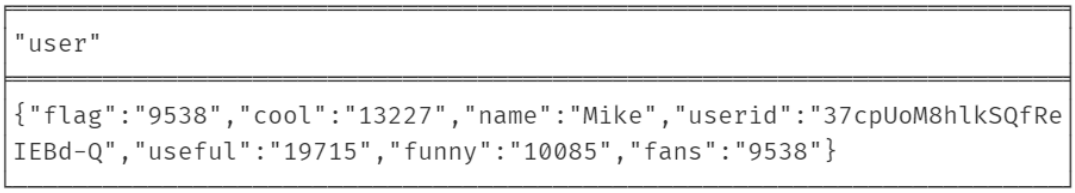


图 4.2　Neo4j 第 16 题查询结果（未添加索引）

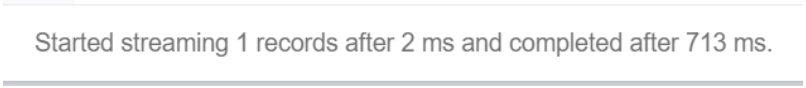


图 4.3　Neo4j 第 16 题查询操作时间（未添加索引）

创建操作的代码如下，用时如图 4.4 所示：

```
MATCH (user:UserNode)    WHERE toInteger(user.fans) = 10000    SET user.flag = 10000
```

图 4.4　Neo4j 第 16 题创建操作时间（未添加索引）

更新操作的代码如下，用时如图 4.5 所示：

```
MATCH (user:UserNode)    WHERE toInteger(user.flag) = 456    SET user.flag = 3456
```

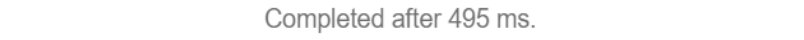


图 4.5　Neo4j 第 16 题更新操作时间（未添加索引）

删除操作的代码如下，用时如图 4.6 所示：

```
MATCH (user:UserNode)    WHERE user.flag > 4000    REMOVE user.flag
```

Set 8 properties, completed after 12 ms.

图 4.6　Neo4j 第 16 题删除操作时间（未添加索引）

然后，重置 flag 属性并为其添加索引，代码如下所示。

```
CREATE INDEX FOR (user:UserNode) ON (user.flag)
```

重复上述四种操作并记录时间，添加索引前后的相应操作时间对比如表 4.1 所示。

表 4.1　Neo4j 第 16 题增删改查时间开销（有索引、无索引对比）

情况/操作用时（ms）	查询	创建	更新	删除
无索引	713	643	495	12
有索引	507	731	584	673

可见，建立索引会对查询操作带来性能提升，但由于需要对索引进行额外的维护，会导致创建、更新、删除等操作变慢。

4.2.3 子任务 3-17

题目要求：查询与用户 user1（userid: tvZKPah2u9G9dFBg5GT0eg）不是朋友关系的用户中和 user1 评价过相同的商家的用户，返回用户名、共同评价的商家的数量，按照评价数量降序排序（查看该查询计划，了解该查询的执行计划及查询执行时间，并给出物理优化手段，以提高查询性能，通过优化前后的性能对比展现优化程度）。

分析与实现：本题可以首先找出 user1 评价过的商家，然后找出所有评价过这些商家的用户，再判断这些用户中哪些与 user1 不是朋友关系。在使用 count 统计数量时，需要注意对商家的 business_id 去重，因为可能存在用户对同一个商家评价多次的情况。代码如下。

```
MATCH (user1:UserNode {userid: 'tvZKPah2u9G9dFBg5GT0eg'})-[:Review]->(:ReviewNode)-[:Reviewed]->(b:BusinessNode)
WITH user1, COLLECT(DISTINCT b) AS user1_businesses
MATCH (user2:UserNode)-[:Review]->(:ReviewNode)-[:Reviewed]->(b:BusinessNode)
WHERE NOT EXISTS ((user1)-[:HasFriend]->(user2)) AND NOT EXISTS ((user2)-[:HasFriend]->(user1)) AND b IN user1_businesses AND user1<>user2
WITH user1, user2, COUNT(DISTINCT b) AS num
RETURN user1.name, user2.name, num
ORDER BY num DESC
```

使用 PROFILE 查看查询计划，如图 4.7 所示。

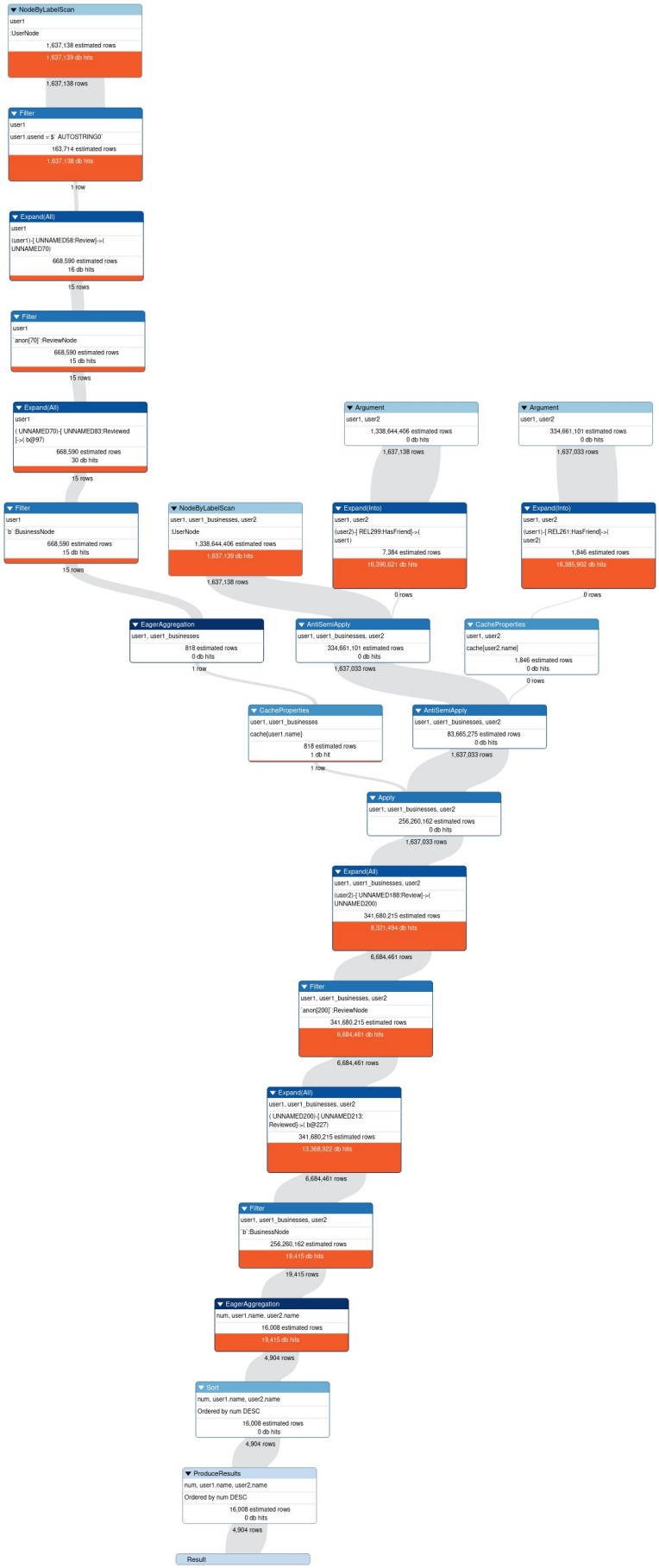
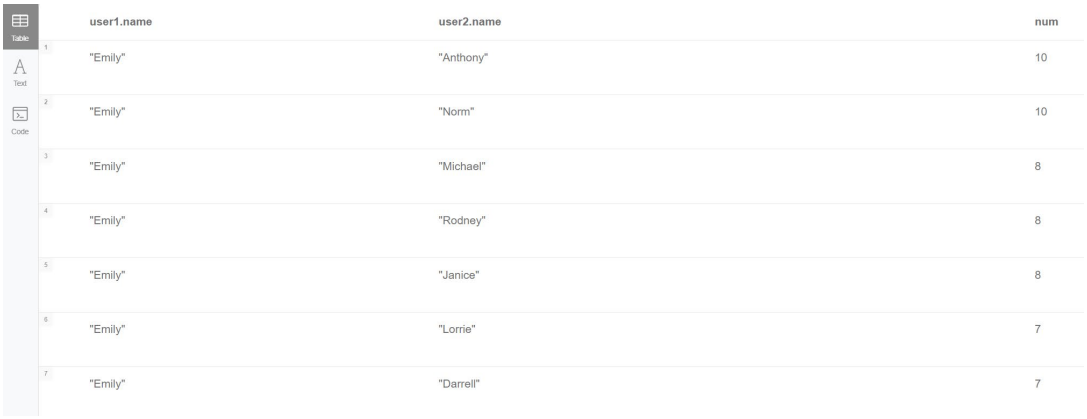


图 4.7　Neo4j 第 17 题查询计划（优化前）

查询结果如图 4.8 所示，去重后的结果与参考结果一致。



	user1.name	user2.name	num
1	"Emily"	"Anthony"	10
2	"Emily"	"Norm"	10
3	"Emily"	"Michael"	8
4	"Emily"	"Rodney"	8
5	"Emily"	"Janice"	8
6	"Emily"	"Lorrie"	7
7	"Emily"	"Darrell"	7

图 4.8　Neo4j 第 17 题查询结果

观察图 4.7 所示的查询计划。首先，对整个用户（UserNode）表做全表扫描，总共 1637138 行数据，从中查找到 userid 满足查询条件的 user1。根据 user1 评价过的商家，查询得到目标商家 BusinessNode 集合，共 15 行数据。接下来，再次全表扫描用户表（需要扫描两次，因为查询代码中涉及到两条有向边的判断），逐一判断与 user1 不是朋友关系的用户集合 user2，最后在 Apply 阶段得到了 1637033 个满足要求的用户。从 user2 集合中，对每一行数据在 EagerAggregation 阶段统计与 user1 评价过的相同商家的数量，最后在 Sort 阶段对结果进行排序，得到的答案共有 4904 行。

通过上述分析，可以做出合理的物理优化。首先，在查询 user1 以及 user2 时遍历了整个用户表，因此可以为 userid 和 businessid 建立索引，通过索引扫描的方式大幅提高查询效率。代码如下所示。

```
CREATE INDEX FOR (user:UserNode) ON (user.userid)
CREATE INDEX FOR (b:BusinessNode) ON (b.businessid)
```

然后将上面的 COLLECT 函数改为仅对 business.businessid 聚合，减少内存开销，如下。

```
WITH user1, COLLECT(DISTINCT b.businessid) AS user1_businesses
```

除了上面两种优化，在判断哪些用户与 user1 不存在朋友关系时，实际上可以只简单地判断 user1 是否有一条指向 user2 的边，或者 user2 是否有一条 user1 的边。这是因为 HasFriend 关系一旦存在就必然是双向的，因此只需要单独判断一个方向即可。

```
WHERE NOT EXISTS ((user1)-[:HasFriend]->(user2)) …
```

修改查询代码后，再次查看查询计划，如图 4.9 所示。

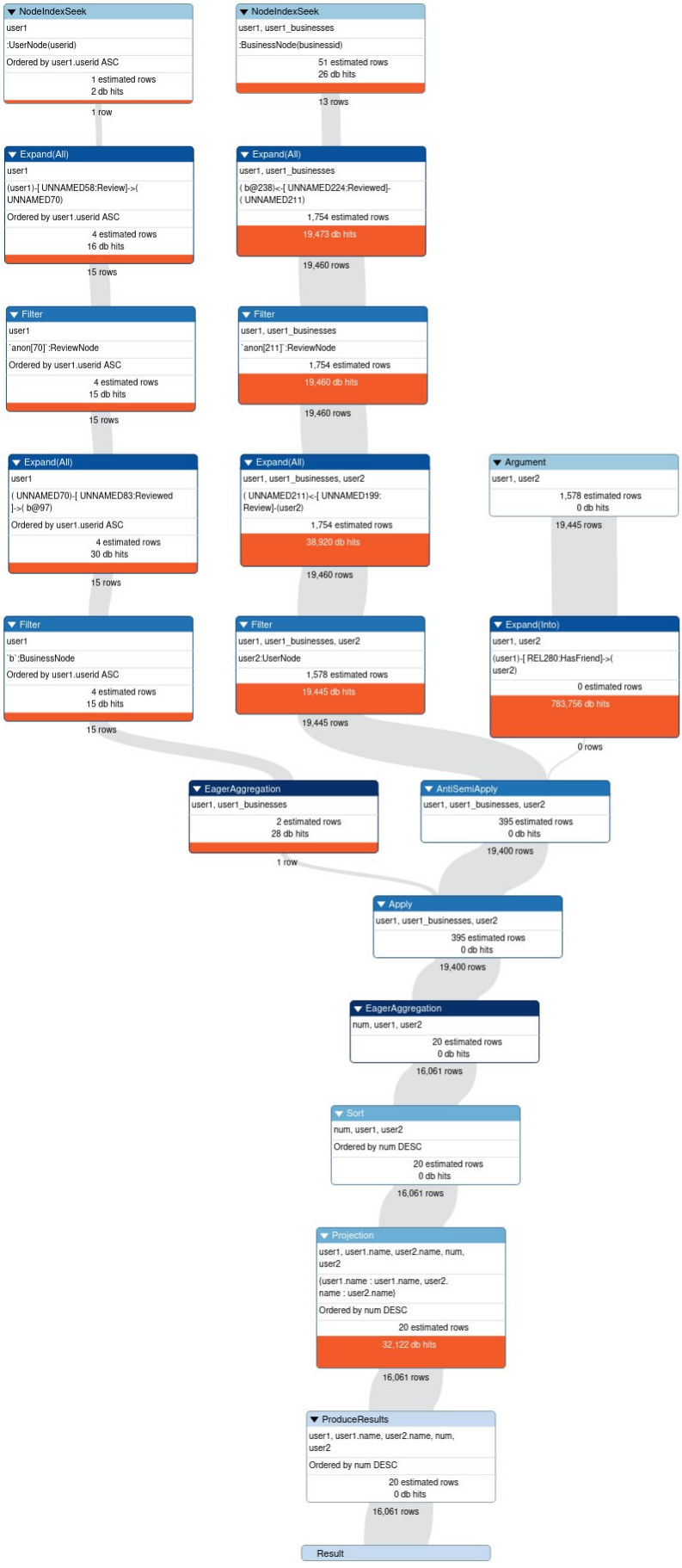


图 4.9　Neo4j 第 17 题查询计划（优化后）

可以看出在优化后的执行计划里，在查找 user1 的过程中通过 NodeIndexSeek 阶段完成了索引的扫描，只产生了 2db hits，相比之前的效率大大提高。同理，由于已经为 businessid 添加了索引，因此查询 user1 评价过的商家节点时也通过扫描索引的方式加快了查询的速度。在查询 user2 集合时，由于查询条件从之前的两条有向边变为一条有向边，因此查询树直接减少了一个分支，避免了大量的重复计算。

优化前和优化后的查询时间分别为 68ms 和 1ms，分别如图 4.10、图 4.11 所示，可以证明优化程度很大。

图 4.10　Neo4j 第 17 题查询用时（优化前）

图 4.11　Neo4j 第 17 题查询用时（优化后）

4.2.4 子任务 3-18

题目要求：使用 Neo4j 和 MongoDB 查询 review_id 为 TIYgnDzezfeEnVeu9jHeEw 对应的 business 信息，比较两者查询时间，指出 Neo4j 和 MongoDB 主要的适用场景。

分析与实现：本题的查询需求很简单，此处不再赘述，重点关注两个不同数据库查询时间的差异。MongoDB 查询所用时间略大于 Neo4j，据此可以指出 Neo4j 和 MongoDB 主要的适用场景。

Neo4j 适用于复杂的图结构关系型数据查询，如社交网络、推荐系统、知识图谱等。由于其使用基于图论的查询语言 Cypher，因此十分擅长图数据的快速查询和分析。它还提供了丰富的图形算法与可视化工具，特别适合于图形分析。

而 MongoDB 适用于海量复杂的非结构化或半结构化数据存储和查询，如日志和文档数据库等，具有高性能、高可扩展和高灵活性的优势。由于其使用了文档数据库模型，因此支持高效的数据插入和查询，以及分布式数据库集群的横向扩展。

4.3 任务小结

在 Neo4j 的实验里，我学会了如何使用 Cypher 查询语言在这一图数据库引擎完成查询、更新、删除等操作，以及使用 PROFILE 查看查询计划并结合实际情况对查询语句进行优化，例如引入索引等。Neo4j 的亮眼之处在于它拥有可视化的图形界面，可以将查询结果形象直观地展示出来，因此在管理一些特定类型的大数据时可以提供很好的帮助。

5　多数据库交互应用实验

5.1 任务要求

实验四主要是在两个或三个数据库之间实现交互应用。例如首先使用 Neo4j 进行查询并导出结果集，再将其导入 MongoDB 进行下一步查询。进一步，还可以将 MongoDB 得到的第二步的结果集再次导入 Neo4j 中完成后续操作。

5.2 完成过程

5.2.1 子任务 4-3

题目要求：在 Neo4j 中查找所有商家，要求返回商家的名字、所在城市、商铺类型：（1）将查找结果导入 MongoDB 中实现对数据的去重（使用 aggregate，仅保留城市、商铺类型）；（2）将结果导入 Neo4j 中的新库 result 中，完成（City-[Has]->Category）图谱的构建。

分析与实现：首先按照题意写出查询代码（在 Neo4j 中完成）。

```
MATCH (business:BusinessNode)-[:IN_CATEGORY]->(c:CategoryNode)
RETURN business.name as name, business.city as city, c.category as category
```

Neo4j 查询完成后，将结果导出为 csv 文件（export.csv），使用下面的命令行操作将其上传到服务器中，如图 5.1 所示。

```
C:\GAP\大数据管理实验>scp ./export.csv root@1.94.55.43:/root/
```

```
C:\GAP\大数据管理实验>scp ./export.csv root@1.94.55.43:/root/
The authenticity of host '1.94.55.43 (1.94.55.43)' can't be established.
ED25519 key fingerprint is SHA256:HO4FMGJ9ay3Di2cnNBp/IZ3bkotVikwwN8BWJLP4NR8.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])?
Warning: Permanently added '1.94.55.43' (ED25519) to the list of known hosts.
root@1.94.55.43's password:
export.csv                                                    100% 2375KB   8.7MB/s   00:00
```

图 5.1　将 Neo4j 查询结果上传服务器

然后启动 Mongo 服务，在 MongoDB 的 yelp 数据集创建一个名为 AllBusiness 的集合。此后退出 mongoDB 并回到主目录，把数据导入到 AllBusiness 集合中，如图 5.2 所示。

```
db.createCollection("AllBusiness")
mongoimport -d=yelp -c=AllBusiness --type=csv --headerline ./AllBusiness.csv
```

```
root@ecs-c925:~# mongoimport -d=yelp -c=AllBusiness --type=csv --headerline ./AllBusiness.csv
2023-10-27T15:40:02.973+0800    connected to: mongodb://localhost/
2023-10-27T15:40:05.973+0800    [##########..........] yelp.AllBusiness     16.1MB/32.3MB (50.0%)
2023-10-27T15:40:08.973+0800    [####################.] yelp.AllBusiness     31.7MB/32.3MB (98.0%)
2023-10-27T15:40:09.090+0800    [####################.] yelp.AllBusiness     32.3MB/32.3MB (100.0%)
2023-10-27T15:40:09.090+0800    788359 document(s) imported successfully. 0 document(s) failed to import.
```

图 5.2　将 csv 文件导入到 MongoDB 集合中

接下来使用 aggregate 对 AllBusiness 去重，仅保留城市、商铺类型，同时需要创建一个名为 DistinctBusiness 的集合用于保存结果，代码如下。

```
db.createCollection("DistinctBusiness")
db.AllBusiness.aggregate([{ $group: { id: { city: '$city', category:
'$category' } } } ]).forEach((item) => { db.DistinctBusiness.insert( item.id ) } )
```

随后按照下面的代码，将结果导出到服务器主目录下的 result.csv 中，然后将其放在 neo4j 安装目录的 import 下。将该结果导入 Neo4j 中的新库 result 中，即可完成（City-[Has]->Category）图谱的构建，如图 5.3 所示。

```
mongoexport -d yelp -c DistinctBusiness --type=csv --fields city,category --out result.csv
cd ~/neo4j-community-4.0.9/import
cp /root/result.csv ./
LOAD CSV WITH HEADERS FROM "file:///result.csv" AS f MERGE (c:CityNode {city:
COALESCE(f.city, "")}) MERGE (a:CategoryNode {category: COALESCE(f.category, "")})
CREATE (c) -[:Has]-> (a)
```

Added 125 labels, created 125 nodes, set 125 properties, created 67536 relationships, completed after 234695 ms.

图 5.3　将去重后的 result.csv 导入到 Neo4j 中

最后查看图谱效果，如图 5.4 所示。

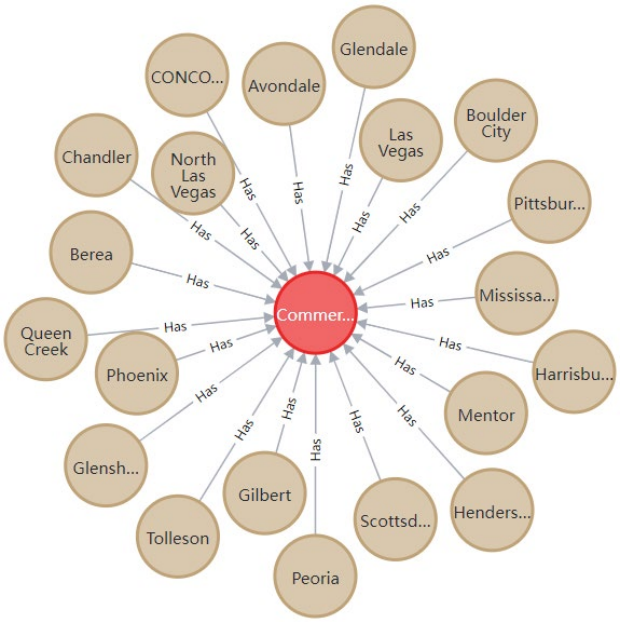


图 5.4　查看构建好的图谱

5.3 任务小结

在本节实验中，我学会了将查询结果从 Neo4j 导出到 MongoDB 再导出到 Neo4j，了解了如何使用多种数据库和查询语言来进行数据操作和关系构建，理解了多数据库之间怎样进行交互。同时，我也意识到了数据去重和图谱构建在实际应用中的价值，对于大规模数据的管理和分析有着重要的作用。

6　不同类型数据库 MVCC 多版本并发控制对比实验

6.1 任务要求

实验五需要分别在 MySQL 和 MongoDB 两种数据库下体验 MVCC 并发控制特性，自行构造用户多写多读样例（用户同时对同一数据库对象的增删改查案例），观察现象并进行对比分析。

6.2 完成过程

6.2.1 子任务 5-1

题目要求：体验 MySQL 在 InnoDB 存储引擎下的 MVCC 多版本并发控制实现的事务 ACID 特性。需要构造多用户多写多读案例展现 MVCC 并发控制特性，并注意 Mysql 需要选用何事务隔离级来支持 MVCC。解释各种结果产生的原因。

分析与实现：首先在 MySQL 中创建一个数据库 testdb 用于测试。在 testdb 中创建一个新表 test，设置其引擎为 InnoDB。

```
create table test (
  id int(10) not null, name varchar(20) not null, flag int(5) not null, primary key(id)
) engine=InnoDB;
```

为了支持 MVCC 特性，需要选用可重复读（Repeatable Read）隔离级别，在该级别下，事务在执行期间会看到一致的数据快照，并防止其它事务对数据进行修改。代码如下所示。

```
set session transaction isolation level repeatable read;
```

插入初始数据并查看当前表内的数据，如图 6.1 所示。设当前会话窗口为 A。

```
mysql> select * from test
    -> ;
+----+------+------+
| id | name | flag |
+----+------+------+
|  1 | LJP  |   27 |
|  2 | DJE  |   42 |
|  3 | OFW  |   61 |
|  4 | SLX  |   15 |
+----+------+------+
4 rows in set (0.00 sec)
```

图 6.1　在 MySQL 中插入初始数据

接下来开始一个事务。然后打开会话窗口 B（直接在 Xshell 中完成）。窗口 A 将 id 为 3 的 flag 更新为 88，与此同时，在另外一边，窗口 B 将 id 为 3 的 flag 更新为 99。

```
start transaction;
```

```
update test set flag=88 where id=3; //窗口 A
update test set flag=99 where id=3; //窗口 B
```

如图 6.2 所示，在窗口 A 中查询当前表内的数据，可以发现 A 对 id 为 3 的数据更新成功，在本地查询的 flag 为 88。但是对于窗口 B，如图 6.3 所示，其对 id 为 3 的数据的更新操作受到阻塞（“Query execution was interrupted”），且在二次查询时发现该数据的 flag 仍然为 A 更新之前的值，也就是说在窗口 A 对数据的更新操作提交（commit）之前，会屏蔽窗口 B 对同一数据的更新操作，同时由于 MVCC 的特性，窗口 B 在查询数据时仍然会使用本地保存的版本。

```
mysql> select * from test;
+----+------+------+
| id | name | flag |
+----+------+------+
|  1 | LJP  |   27 |
|  2 | DJE  |   42 |
|  3 | OFW  |   88 |
|  4 | SLX  |   15 |
+----+------+------+
4 rows in set (0.00 sec)
```

图 6.2　MySQL 用户多写多读案例（窗口 A）

```
mysql> update test set flag=99 where id=3;
^C^C -- query aborted
ERROR 1317 (70100): Query execution was interrupted
mysql> select * from test;
+----+------+------+
| id | name | flag |
+----+------+------+
|  1 | LJP  |   27 |
|  2 | DJE  |   42 |
|  3 | OFW  |   61 |
|  4 | SLX  |   15 |
+----+------+------+
4 rows in set (0.00 sec)
```

图 6.3　MySQL 用户多写多读案例（窗口 B）

现在，再在窗口 A 中完成提交 commit 操作，到窗口 B 中查询当前数据，发现此时的数据变为最新版本。此时窗口 B 可以正常对 id 为 3 的数据进行更新，如图 6.4 所示。

```
mysql> select * from test;
+----+------+------+
| id | name | flag |
+----+------+------+
|  1 | LJP  |   27 |
|  2 | DJE  |   42 |
|  3 | OFW  |   88 |
|  4 | SLX  |   15 |
+----+------+------+
4 rows in set (0.00 sec)

mysql> update test set flag=99 where id=3;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

图 6.4　MySQL 用户多写多读案例（窗口 B，当窗口 A 提交后）

6.2.2 子任务 5-2

题目要求：体验 MongoDB 的 MVCC，主要是通过创建 MongoDB 分片集群实现，数据集可自建，测试方法同上。需要对测试结果进行说明，并与 MySQL 的 MVCC 实验结果进行对比分析。

分析与实现：首先需要完成分片的创建。在华为云平台上创建三台服务器（test-0001、test-0002、test-0003），其弹性公网 ip 分别为：60.204.242.167、60.204.228.189、60.204.244.116。创建完成后的架构如图 6.5 所示。

图 6.5　MongoDB 分片集群架构

分别完成 shard1 和 shard2 的 replica set 配置并初始化，如图 6.6 所示。随后按照部署手册方法完成 config server 和 mongos 的配置。

```
root@test-0001:/usr/local/mongodb# mongod --shardsvr --replSet shard1 --port 27017 --dbpath /usr/loc
al/mongodb/data/shard11 --oplogSize 2048 --logpath /usr/local/mongodb/data/shard11.log --logappend -
-bind_ip=0.0.0.0 --fork
about to fork child process, waiting until server is ready for connections.
forked process: 8502
child process started successfully, parent exiting
```

图 6.6　Replica set 的配置与初始化

接下来即可使用 mongos。切换到 admin，并添加分片，如图 6.7 所示。

```
mongo 60.204.242.167:30000
use admin;
db.runCommand({addshard:"shard1/60.204.242.167:27017,60.204.228.189:27017,60.204.244.116
:27017",name:"s1", maxsize:20480});
db.runCommand({addshard:"shard2/60.204.242.167:27018,60.204.228.189:27018,60.204.244.116
:27018",name:"s2", maxsize:20480});
```

```
root@test-0001:/usr/local/mongodb# mongo 60.204.242.167:30000
MongoDB shell version v4.4.17
connecting to: mongodb://60.204.242.167:30000/test?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("d7addd2e-c143-4e61-87f5-45ff525820b8") }
MongoDB server version: 4.4.17
---
The server generated these startup warnings when booting:
        2023-10-30T20:44:18.813+08:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
        2023-10-30T20:44:18.813+08:00: You are running this process as the root user, which is not recommended
---
mongos> use admin;
switched to db admin
mongos> db.runCommand({addshard:"shard1/60.204.242.167:27017,60.204.228.189:27017,60.204.244.116:27017",name:"s1", maxsize:20480});
{
        "shardAdded" : "s1",
        "ok" : 1,
        "operationTime" : Timestamp(1698669951, 5),
        "$clusterTime" : {
                "clusterTime" : Timestamp(1698669951, 5),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
}
mongos> db.runCommand({addshard:"shard2/60.204.242.167:27018,60.204.228.189:27018,60.204.244.116:27018",name:"s2", maxsize:20480});
{
        "shardAdded" : "s2",
        "ok" : 1,
        "operationTime" : Timestamp(1698669965, 3),
        "$clusterTime" : {
                "clusterTime" : Timestamp(1698669965, 3),
                "signature" : {
                        "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
                        "keyId" : NumberLong(0)
                }
        }
}
mongos>
```

图 6.7　mongos 成功添加分片

在 testdb 数据库中创建一个新的集合 testc，插入与 6.2.1 节相同的初始数据（此处不再赘述）。此时打开 test-0002 服务器并进入 mongos，并删除 id 为 3 的数据。

```
mongo 60.204.228.189:30000
db.getSiblingDB("testdb").getCollection("testc").deleteOne({ _id: 3 })
```

然后再回到 test-0001 服务器，通过 db.testc.find()查看当前集合中的数据，如图 6.8 所示，可以发现对数据的删除直接作用于 test-0001，它查询到的数据版本是最新的。这证明 MongoDB 的 MVCC 机制是非阻塞式的。

```
mongos> db.testc.find()
{ "_id" : 1, "name" : "LJP", "age" : 27 }
{ "_id" : 2, "name" : "DJE", "age" : 42 }
{ "_id" : 4, "name" : "SLX", "age" : 15 }
```

图 6.8　MongoDB 用户多写多读案例（test-0001 服务器）

6.3 任务小结

本小节实验主要遇到的问题是在添加 MongoDB 分片时，由于 addshard 字段判定很严格，其中不能出现任何空格，因此在第一次配置时在服务器上造成了不可逆的错误配置，导致我重新购买了三台服务器并重复了所有的配置操作。所幸在注意到这个问题后，很快就配好了分片集群环境。

通过本次实验，我对 MySQL 和 mongoDB 两种数据库的 MVCC 特性进行了测试与对比分析。MySQL 和 MongoDB 都实现了 MVCC（多版本并发控制）机制，用以解决读写冲突的并发控制。在 MVCC 机制中，为事务分配单向增长的时间戳，并为每个修改保存一个版本，版本与事务时间戳关联。读操作只会读取该事务开始前的数据库快照，从而避免阻塞其它读操作。

虽然二者都采用了 MVCC，但在具体实现上存在显著差异。MySQL 的 MVCC 是通过保存数据的历史版本来实现的，每个数据项都有一个时间戳 timestamp，可以追踪到创建和修改的时间点。这种实现方式使得 MySQL 能够提供严格的可重复读，保证读取操作不会看到未提交（commit）的修改，对已修改但未提交的数据进行更新也是非法的。

MongoDB 的 MVCC 实现更复杂。在 MongoDB 中，每个文档都有一个包含多个版本的时间戳数组，也可以看作一个 topologyVersion（拓扑版本号）。当文档被修改时，旧版本并不会被删除，而是存储在数组中，而在绝大部分情况下读写操作都使用的是当前最新的版本（当然，这种方式也允许 MongoDB 在读取数据时查看过时的数据版本），从而实现非阻塞读写操作。

7 课程总结

本学期的大数据管理实验主要包括 Mysql for Json 实验、MongoDB 实验、Neo4j 实验、多数据库交互应用实验和 MVCC 多版本并发控制对比实验。通过这些实验，我对不同数据库的特点和使用方法有了更深入的了解。

在 Mysql for Json 实验中，我在云服务器上配置了 Mysql 环境，并成功下载了数据集。在实验中，我熟悉了基本查询语句与聚集函数的使用技巧。通过使用 explain 函数，我得以查看到数据库对查询语句的具体执行计划，帮助我对其进行分析和优化。

在 MongoDB 实验中，我了解到索引对查询的帮助很大，特别是在数据条数很多时，通过为合适的数据属性添加索引可以显著降低查询开销。我还成功实现了一个 MapReduce 程序，可以在多个节点上并行处理提高数据处理效率。

在 Neo4j 实验中，我学会了使用 Cypher 语言完成查询、更新、删除等操作，以及使用 PROFILE 查看查询计划并进行优化。Neo4j 的可视化图形界面给数据展示提供了很大的帮助，尤其在管理特定类型的大数据时更能发挥其优势。

在多数据库交互应用实验中，我尝试将查询结果从 Neo4j 导出到 MongoDB，再导出到 Neo4j，实现了借助多种数据库和查询语言的数据操作和关系构建。

最后，我通过构造实际的用户多写多读案例，对 MySQL 和 MongoDB 两种数据库的 MVCC 特性进行了测试与对比。其中，MySQL 的 MVCC 通过保存数据的历史版本来实现，而 MongoDB 的 MVCC 实现更复杂，每个文档都有一个时间戳数组来存储多个版本。最终可以得出结论，MySQL 能够提供严格的可重复读，而 MongoDB 则实现了非阻塞读写操作。

通过本次实验，我进一步深入学习和掌握了各种数据库的高级功能和优化技巧。在日后的学习中，我希望能结合现实中的实际需求，深入研究和实践多数据库之间的数据交互，使大数据管理变得更加高效且实用。